IPDS '98
Final Version.

Coordinated Checkpointing Without Direct Coordination

Nuno Neves
Coordinated Science Laboratory
Univ. of Illinois at Urbana-Champaign
Urbana, IL 61801

W. Kent Fuchs
School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN 47907

Abstract

Coordinated checkpointing is a well-known method to achieve fault tolerance in distributed systems. Long running parallel applications and high-availability applications are two potential users of checkpointing, although with different requirements. Parallel applications need low failure-free overheads, and high-availability applications require fast and bounded recoveries. In this paper, we describe a new coordinated checkpoint protocol capable of satisfying both types of applications. The protocol uses time to avoid all types of direct coordination (e.g., message exchanges and message tagging), reducing the overheads to almost a minimum. To ensure that rapid recoveries can be attained, the protocol guarantees small checkpoint latencies. The protocol was implemented and tested on a cluster of workstations connected by a 155 Mbit/sec ATM. Experimental results show that the protocol overheads are very small.

1. Introduction

Coordinated checkpointing and rollback recovery is a technique for fault-tolerance in distributed systems [25, 10, 11, 4, 12]. During the failure-free periods, a coordinated checkpoint protocol stores periodically in stable storage the state of the application and the messages that are in-transit in the network. When a failure occurs, recovery involves rolling back the application to the last available state, and then restarting its execution. One of the main advantages of coordinated checkpointing, when compared with other checkpointing methods, is its simplicity. For instance, log-based checkpoint protocols have to save the reception order and the contents of all messages, and then must garbage collect this information [23, 28, 8, 21, 1, 20]. Other positive aspects of coordinated checkpointing are that concurrent failures can be tolerated, and applications do not have to execute in a piece-wise deterministic manner [22].

Arguments commonly used against coordinated checkpointing have been the overhead and lack of independence of checkpoint creation. Traditionally, two types of coordination have been employed: extra message exchanges and message tagging. A coordinated protocol sends extra messages, for example, to guarantee that all processes begin to save their states [2, 24]. One reason for using message tagging is to minimize the number of processes that have to roll back after a failure [19]. The main concern of high-availability applications is to minimize down time [7]. With coordinated checkpointing, larger checkpoint periods result on average in larger rollbacks, and consequently in longer recovery times. Typically, a coordinated protocol only begins to create a new application checkpoint when the previous one has been completely stored. Therefore, the checkpoint period is lower bounded by the *checkpoint latency*, which is the time a protocol takes to save a new application checkpoint [27]. Any protocol that has a checkpoint latency dependent on the message delivery times (e.g., uses message coordination) should not use small checkpoint periods. When there is network congestion, message delivery times become much larger than the average values, resulting in checkpoint latencies also larger than the average.

In this paper, we describe a new coordinated protocol that uses time to avoid all types of direct coordination. Although processes save their states independently, the protocol is able to store application checkpoints that are consistent and recoverable. The protocol is specified using two procedures, one that saves the process state whenever a local timer expires, and another that keeps timers approximately synchronized (with reasonably good clocks, timers only need to be resynchronized approximately once a day). The second procedure is also used to detect failures in the processor clocks that might lead to incorrect behavior of the protocol. The protocol also has a checkpoint latency completely independent of message delivery times, enabling it to function correctly even with small checkpoint periods.

Nuno Neves was supported in part by the Grant BD-3314-94, sponsored by the program PRAXIS XXI, Portugal. This research was supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract DABT 63-96-C-0069, and in part by the Office of Naval Research under contract N00014-97-1-1013.

PURDUE UNIVERSITY

SCHOOL OF ELECTRICAL AND
COMPUTER ENGINEERING
WEST LAFAYETTE, IN 47907-1285 USA

W. KENT FUCHS
HEAD

July 7, 1998

Colin E. Wood
ONR 312
Office of Naval Research
Ballston Centre Tower One
800 North Quincy Street
Arlington, VA 22217-5660

Dear Dr. Wood:

Enclosed is a copy of the final version of the paper listed below concerning research supported in part by the Department of the Navy and managed by the Office of the Chief of Naval Research under Grant N00014-95-1-1049:

> N. Neves and W. K. Fuchs, "Coordinated Checkpointing Without Direct Coordination," to appear in the *Proceedings of the International Computer Performance and Dependability Symposium,* Durham, NC, September 1998.

This material is covered under Distribution Statement A (Approved for Public Release: Distribution is Unlimited).

We appreciate your support of this research.

Sincerely,

W. Kent Fuchs
Head and Professor
Electrical and Computer Engineering

Enclosure: manuscript

cc: Administrative Grants Officer (Chicago IL)
Director, Naval Research Laboratory (Washington DC)
Defense Technical Information Center (Ft. Belvoir VA)

The main contributions of this paper are: first, it describes a new protocol with weaker requirements on the support system, when compared with previously proposed time-based protocols [26, 4, 12]. By removing the assumption of small bounded message delivery times, the protocol can be used in systems connected by most types of networks (including ethernet and ATM). Second, the paper presents an improved timer resynchronization procedure that is able to detect clock failures. Third, experimental results are shown for a cluster of workstations connected by a 155 Mbit/sec ATM. The results indicate that the protocol introduces very small overheads.

2. Related Work

Several authors have proposed coordinated checkpoint protocols in the past. Most of these protocols exchange extra messages to coordinate the creation of new checkpoints [25, 10, 11, 18, 9, 2, 5]. More recently, time-based protocols were introduced that rely on approximately synchronized clocks or timers to avoid message coordination [26, 4, 12]. Time-based protocols save checkpoints periodically, whenever local timers expire.

Tong et al. [26] proposed the first time-based protocol. This protocol assumes loosely synchronized processor clocks and relatively small message delivery times. Processes use a positive acknowledgment retransmission scheme to be able to communicate reliably. A process starts to save its state whenever the local clock reaches a multiple of the checkpoint period. The checkpoint of a process includes all messages that have been sent and have not been acknowledged. The protocol adds to each application message and acknowledgment a checkpoint number to detect in-transit messages. In-transit messages are stored in stable storage as they are received. Processor clocks are resynchronized periodically. Cristian and Jahanian [4] also proposed a protocol that uses time to initiate the creation of the checkpoints. This protocol requires stricter assumptions about the synchronization of the clocks and assumes that message delivery times are small with high probability. Like the previous protocol, it tags the application's messages, and saves the in-transit messages at the receiver. Neves and Fuchs [12] proposed a protocol that used synchronized timers instead of synchronized clocks. The protocol assumes small and bounded message delivery times. Processes save their states whenever the local checkpoint timer expires. Contrary to the previous time-based protocols, the protocol prevents the existence of in-transit messages. This is accomplished by disallowing message sends during a certain interval before the checkpoint creation. The size of this interval is proportional to the maximum message delivery time.

The protocol proposed in this paper also uses synchronized timers instead of synchronized clocks. This characteristic is important in systems where, for security reasons, the applications are not allowed to change the value of the processor clocks. The new protocol, however, does not have to prevent the processes from sending messages during an interval of time. Instead of avoiding the existence of in-transit messages, the protocol includes them in the sender's checkpoints. The protocol also does not need the assumption that message delivery times are small. This assumption is common to all previous time-based protocols and is the most important limitation to the applicability of the protocols. The protocol does not tag the application's messages with any information, and it does not make extra accesses to stable storage to store the in-transit messages. Our new approach also guarantees small and bounded checkpoint latencies, completely independent of the message delivery times.

3. System Description

An application is executed by a set of processes running on several nodes. A node contains a processing unit, local memory and a local hardware clock. Clocks do *not* have to be synchronized, however, they drift from real time with a maximum drift rate, ρ. Therefore, during a real-time interval [s,e], a clock accumulates at most an error of $\rho(e-s)$ time units ($(1-\rho)(e-s) \leq C(e)-C(s) \leq (1+\rho)(e-s)$, where $C(t)$ is the value of the clock at the real time t). Processes can set timers to schedule future executions of operations. A timer started at real time t with an initial value T expires at time $C(v) \geq C(u) = C(t)+T$. It is assumed that a process does not execute the program during the interval $[u, v]$, which is called the *scheduling delay*. This interval is limited by a constant δ, and it accounts for the situations when the operating system suspends a process until it finishes another task. If two timers are started in two nodes exactly at the same time with the same initial value T, then they will expire at most $(2\rho T + \delta - \delta\rho^2)/(1-\rho^2)$ time units from each other. This value will be approximated by $2\rho T + \delta$ because drift rates are very small. For most quartz clocks available in commodity workstations, the drift rates are in the order of 10^{-5} or 10^{-6}, and for high precision clocks ρ is in the order of 10^{-7} or 10^{-8} [3]. The value of the scheduling delay is normally in order of tens of milliseconds, however, we will only require that δ is smaller than the checkpoint period.

Processes communicate by exchanging messages. Messages may be lost while in transit, arrive out of order, be duplicated, or may be discarded because of insufficient buffering space. To guarantee in-order reliable message delivery, the application utilizes a communication protocol. Typically, the communication protocol keeps a copy of each sent message until an acknowledgment arrives. If the acknowledgment is lost or delayed, it retransmits the messages after a timeout interval. After retransmitting the message a number of times, if no acknowledgment is received, the protocol assumes that the remote process has failed and returns an error that can be used to initiate recovery [13]. The communication protocol associates with each message a sequence number. When a message is received, it compares the associated sequence number with the expected sequence number to identify communication problems (e.g., duplicates).

It is assumed that it is possible to save, at checkpoint time, a copy of the messages that have not yet been acknowledged and the current values of the send and receive sequence number counters. This information can be easily obtained if the application is built on top of unreliable communication channels (e.g., sockets over UDP). In this case, the reliable communication protocol is implemented as part of the application, which means that the unacknowledged messages and sequence counters are automatically stored when processes create their checkpoints. On the other hand, if the application uses reliable communication channels (e.g., sockets over TCP), the checkpoint protocol has to be able to extract the necessary information. In this case, the checkpoint protocol might have to be implemented together with the communication protocol (normally in the operating system), or it will have to replicate some of the functionality of the communication protocol.

4. Consistent and Recoverable Checkpoints

A coordinated checkpoint protocol periodically saves global states of an application. A global state includes the state of each process that is executing the application, and possibly, some messages. To recover the application from failures, the protocol rolls back all processes to the last global state that is available, and then it lets processes reexecute their programs. Recovery is only correct if the external results of the application reexecution are equivalent to one of the possible results of a failure-free execution [2]. To guarantee that correct recovery is always possible, the protocol should save global states that verify the following two properties:

Consistency: If *rcv(mi)* is reflected in the global state, then *send(mi)* must also be reflected in the global state.

Recoverability: If *send(mi)* is reflected in the global state, then *rcv(mi)* must also be reflected in the global state **or** the checkpoint protocol must be able to restore message *mi*.

Global states are only acceptable for recovery if they correspond to an application state that might have occurred in a failure-free execution. The consistency property avoids rollbacks to global states where a process state reflects the reception of a message that was not sent by another process. This type of global state can appear if a process receives a message before creating its checkpoint, but the message was sent after the sender process stored its state. The recoverability property ensures that all messages that are in-transit in the network at checkpoint time are logged by the protocol. Otherwise, in case of a failure, these messages become lost. The protocol must explicitly store the in-transit messages so that processes can reread them during recovery.

5. Time-Based Checkpointing

The time-based protocol is implemented using two procedures, one that saves the processes' states, and another that keeps the checkpoint timers approximately synchronized. The create checkpoint procedure is kept as simple and efficient as possible since it is executed much more frequently than the resynchronization procedure. Its responsibilities consist mainly in storing a new checkpoint of the process, and in checking the timers to see if they need to be resynchronized. The other procedure is executed infrequently, and serves two purposes: it sets the timers with a deviation smaller than D time units, and it detects failures in the processor clocks that might lead to incorrect behavior of the protocol.

5.1. Checkpoint Creation Procedure

The protocol uses time to indirectly coordinate the creation of new global states. Processes determine the instants when they should save their states using a local timer without needing to exchange messages. By keeping the timers approximately synchronized, the protocol ensures that the independently created checkpoints form a consistent and recoverable global state. In this section, we will assume that timers are initially synchronized in such a way that they expire at most D time units apart.

The protocol has to guarantee that processes create their checkpoints in such a way that the consistency property is verified. This property is satisfied if no process, after storing its checkpoint, sends a message that is read before the receiver saves its state. If timers were exactly synchronized, all processes would initiate their checkpoints at exactly the same time, and no consistency problems could occur. In a distributed system, however, timers are never perfectly synchronized, and they expire within an interval DEV. Therefore, one way to avoid consistency violations consists in guaranteeing that no messages are sent during DEV. If the checkpoint period is T, and n checkpoints have been created since the last timer re-synchronization, then the interval DEV is bounded by a *maximum deviation* $MD = D + 2\rho nT + \delta \geq DEV$. The value of MD is composed of three factors: the initial deviation among the timers D; the drift of the clocks $2\rho nT$; and the scheduling

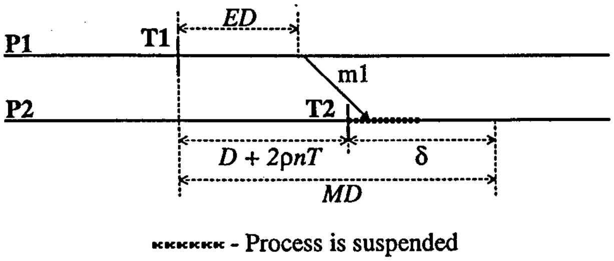

Figure 1. Consistency condition.

```
procedure createCkp()
1  saveProcessState();
2  $N = N + 1$;
3  $ckpTime = ckpTime + T$;
4  setTimer($createCkp, ckpTime$);
5  if (($D + 2\rho(N-1)T - td_{min}$) >
           (getTime() − ($ckpTime - T$)))
6      requestResyncTimers();
```

Figure 2. Checkpoint creation procedure.

delay δ. Processes, however, do not know if their timers expire at the beginning or at the end of the interval MD. One approach to address this problem consists of preventing message sends during MD time units after the timers expire.

In practice, the interval MD is too conservative for two reasons: first, since messages take a minimum time to be delivered, td_{min}, processes can start to send messages earlier without causing consistency violations. A message transmitted at the end of the interval $MD - td_{min}$ will arrive later than all timers expired, and consequently after the receiver process has initiated its checkpoint. Second, since it is assumed that processes do not execute during the scheduling delay, messages that arrive during this period will not be read. Therefore, it is possible to define a tighter interval where messages should not be transmitted. This interval is called *effective deviation* and is equal to $ED = D + 2\rho nT - td_{min}$.

Figure 1 shows the execution of two processes with checkpoint timers scheduled to stop at $T1$ and $T2$. The maximum interval that separates $T1$ from $T2$ is equal to $D + 2\rho nT$. Due to the scheduling delay, the timer of process $P2$ expires later than $T2$. When process $P2$ starts to execute, it creates a new checkpoint, and only after it reads message $m1$, avoiding the consistency problem.

Typical communication protocols, either implemented as part of the application or in the operating system, ensure reliable message deliveries by keeping a copy of each sent message until an acknowledgment arrives. Lost or corrupted messages are recovered by re-transmitting the messages if the acknowledgment is not received within a given interval, and duplicate messages are detected using sequence numbers. An in-transit message is a message that was sent before the sender process saved its state, and is received after the receiver process created its checkpoint. Therefore, unless acknowledgments violate the consistency property, a copy of each in-transit message exists in the sender machine at checkpoint time.

A checkpoint protocol can ensure that all in-transit messages are logged by including the unacknowledged messages in the sender checkpoint. During recovery, the sender re-transmits the logged messages, avoiding message losses. Logging at the sender, however, might save a few other messages besides the in-transit messages; a process can receive a message before its checkpoint, but the acknowledgment might only arrive after the sender has started to save its state (see example in Section 5.3). These extra messages have to be detected and discarded during recovery, otherwise they are read twice. This can be accomplished by saving, in the checkpoints, the value of the send and receive sequence number counters, which are used to detect duplicate messages due to re-transmissions. By resetting these counters during recovery, the extra messages will be considered duplicated messages and will be removed automatically.

The code from Figure 2 can be used to implement the checkpoint creation procedure. The procedure starts by saving the process state and by preparing the timer for the next checkpoint (Lines 1-4). The function `saveProcessState()`, stores the process state in stable storage, including all unacknowledged messages and the send and receive sequence number counters. Next, `createCkp` tests if ED seconds have passed since the checkpoint time (Line 5). If the condition is not satisfied, this means that the term $2\rho nT$ has grown too large, and that timers need to be re-synchronized. The frequency of resynchronization, however, is usually small. For reasonably good clocks ($\rho = 10^{-6}$), checkpoint sizes of 1 MBytes, and stable storage with bandwidth of 5 MBytes/s, the resynchronization procedure only needs to be run once a day.

5.2. Resynchronization Procedure

The time-based protocol utilizes the resynchronization procedure to keep the checkpoint timers approximately synchronized, and to detect failures in the clocks that might result in incorrect behavior of the protocol. In this section, whenever we say that "a timer expires at a certain time", we will be referring to the instant that the timer is scheduled to stop.

The resynchronization procedure selects one of the pro-

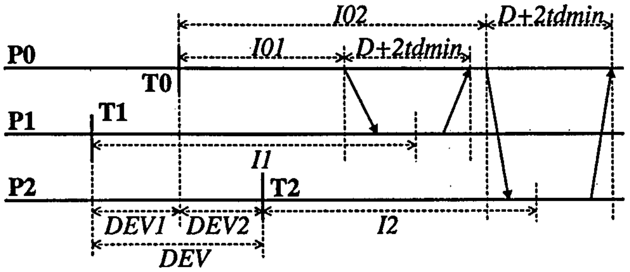

Figure 3. Estimate the value of DEV.

cesses to act as coordinator. The coordinator adjusts the other processes' timers in such a way that they expire at most D time units apart from its timer (if ρ and δ are zero). Since clocks might not be synchronized, it can not ask another process to set the timer to a given absolute time. Instead, it uses a simple iterative method; first, the coordinator sends to the other process the interval until the next checkpoint, $interval_c = ckpTime_c - currentTime_c$. Second, the process saves the $interval_c$ and the current time, $currentTime_p$, and sends an acknowledgment back to the coordinator. Third, if the time that separates the transmission of $interval_c$ and the reception of the acknowledgment is smaller than $D + 2td_{min}$ time units, the coordinator sends an END synchronization message. Otherwise, it returns to the first step and repeats the same operations. When the remote process receives the END message, it resets the timer with the new checkpoint time $ckpTime_p = currentTime_p + interval_c - td_{min}$.

The resynchronization method has to be done in an iterative way because delivery times are not constant. In some cases the first message of the coordinator can take more time to arrive than in others. Therefore, the coordinator can only guarantee a deviation smaller than D if it rejects all iterations where the round trip time is larger than $D + 2td_{min}$. Nevertheless, messages usually have short delivery times, which allows D to be kept small. The constant D should be made equal to a multiple of the round trip-time of a small message. As a rule of thumb, we normally set D to 10 ms, which works well for a 10/100 Mbit/s Ethernet and 155 Mbit/s ATM networks.

The time-based protocol is only able to ensure that global states are consistent if timers expire within the expected deviation, $DEV_{expected} = D + 2\rho nT$. For this reason, it is important to detect clock failures. While the coordinator adjusts the timers, it estimates the maximum deviation between two timers during the last checkpoint creation, DEV. If DEV is larger than $DEV_{expected}$, then at least one of the clocks is not working correctly.

The example from Figure 3 is used to illustrate the calculation of DEV. From the three processes represented in the figure, $P0$ is the coordinator. $T0$, $T1$, and $T2$ were the times when the timers were scheduled to end during the last checkpoints. The coordinator estimates the value of $DEV1$ while it resynchronizes the timer of process $P1$. Before sending the $interval_c$, the coordinator calculates the interval $I01$. After receiving $interval_c$, process $P1$ computes the value of $I1$ and returns it to the coordinator in the acknowledgment. If the coordinator accepts the resynchronization, $DEV1$ can be estimated in the following way:

$$|\Delta_1| - D \leq DEV1 \leq |\Delta_1| + D,$$
$$with\ \Delta_1 = (I01 + td_{min}) - I1$$

Using an equivalent method, the coordinator obtains the estimate of $DEV2$. Next, it computes DEV using the estimates $DEV1$ and $DEV2$. Two cases have to be considered: both timers expired before or after $T0$, or one of the timers expired before and another after $T0$. The following bounds can be derived for DEV,

$$\begin{cases} max(|\Delta_1|, |\Delta_2|) - D \leq DEV \leq max(|\Delta_1|, |\Delta_2|) + D \\ \qquad if\ ((\Delta_1 < 0)\ and\ (\Delta_2 < 0)) \\ \qquad or\ ((\Delta_1 > 0)\ and\ (\Delta_2 > 0)) \\ |\Delta_1| + |\Delta_2| - 2D \leq DEV \leq |\Delta_1| + |\Delta_2| + 2D \\ \qquad otherwise \end{cases}$$

The coordinator can make the following conclusions by comparing the expected deviation with the estimated maximum and minimum deviations:

$$\begin{cases} DEV_{expected} > DEV_{max} & OK \\ DEV_{expected} < DEV_{min} & Failure \\ DEV_{min} \leq DEV_{expected} \leq DEV_{max} & No\ conclusion \end{cases}$$

Since DEV is not exactly determined, there is a window of uncertainty of size $4D$ time units where conclusions can not be made about the failure of the processor clocks. A pessimistic or optimistic approach can be used to address this problem; with the pessimistic approach, the protocol assumes that there was a failure if $ED_{expected} \leq DEV_{max}$. The optimistic approach uses the condition $ED_{expected} < DEV_{min}$ to detect clock failures. If a clock failure is found then a system warning can be issued and the protocol can increase the assumed value for the drift rate (e.g., $\rho_{new} = \rho_{old} * 2$). If, after a few resynchronizations, the value of ρ does not converge to the *real* drift rate, then another warning must be sent saying that the protocol is unable to recover from the clock failure.

The resynchronization procedure is implemented using the code from Figure 4. The code adjusts the checkpoint timers and detects the clock failures. For this reason, it should only be utilized after processes have created at least one checkpoint. The first synchronization can be done using an equivalent procedure with the lines corresponding

```
procedure ResynchronizeTimers()
   Coordinator:
1  $\Delta_a = \Delta_b = 0$;
2  for each($p \in Processes$) do
3    while (TRUE) {
4      $currentTime_c$ = getTime();
5      send($p, ckpTime - currentTime_c$);
6      receive($p, I$);
7      if ((getTime() $- currentTime_c) < (D + 2 * td_{min})$)
8      {
9        $I0 = (currentTime_c + td_{min}) - (ckpTime - T)$;
10       if ($I > I0$) $\Delta_b$ = max($\Delta_b, I - I0$);
11       else $\Delta_a$ = max($\Delta_a, I0 - I$);
12       break;
13     }
14   }
15 broadcast($-1$);
16 if(($\Delta_a + \Delta_b + 2D) \geq (D + 2\rho(N-1)T)$) Error();
17 $N = 1$;

   Process p:
18 receive($coord, loop$);
19 do {
20   $currentTime_p$ = getTime();
21   $interval_c = loop$
22   send($coord, currentTime_p - (ckpTime - T)$);
23   receive($coord, loop$);
24 } while ($loop > 0$);
25 $ckpTime = currentTime_p + interval_c - td_{min}$;
26 setTimer($createCkp, ckpTime$);
27 $N = 1$;
```

Figure 4. Timer resynchronization procedure.

to the fault detection removed. The coordinator executes the `while` loop to synchronize the remote timers (Lines 2-3). It begins by sending the $interval_c$, next waits for the acknowledgment, and then sees if the round-trip time is smaller than $D + 2td_{min}$ (Lines 4-7). In the affirmative case, it breaks the loop and begins to set another timer (Line 12). The END message is sent when all timers have been adjusted (Line 15). The code extends the fault detection method described previously to a number of processes larger than three. The coordinator estimates the two deviations by receiving the interval from the other process (Lines 1, 6, and 9-11). The pessimistic approach is used to determine if there was a clock failure (Line 16). The other processes receive $interval_c$ (Lines 18-24), and then they reset the timers (Lines 25-26).

5.3. Examples

The example from Figure 5 shows the execution of the protocol during the creation of an application checkpoint. The two processes begin to save their states at different instants because checkpoint timers are not exactly synchronized. However, since timers expire at most *MD* seconds apart, the two checkpoints are separated by a small interval. Message *m1* is not logged by the protocol since at *T2* the acknowledgment has already arrived. Both messages *m2* and *m3* are logged by the protocol. Message *m2* is sent by process *P1* and is received by process *P2* before the creation of the checkpoints meaning that *m2* is not an in-transit message. However, the acknowledgment of *m2* only arrives to process *P1* after the timer expires. At *T1*, process *P1* does not know if *m2* is an in-transit message or not, so it includes the message in the checkpoint.

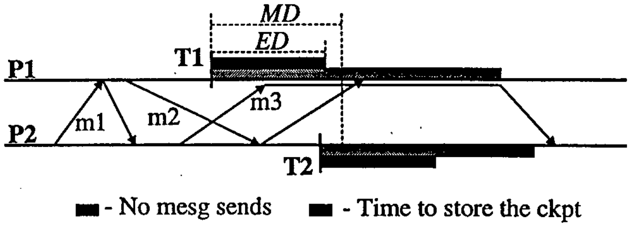

Figure 5. Creation of a checkpoint.

Later, if there is a failure, processes will have to roll back to the stored checkpoints, and process *P1* will resend *m2*. Process *P2* will detect that *m2* is a duplicate using the receive sequence number counter that was included in its checkpoint, and will remove the message. (Although the acknowledgment of *m2* is an in-transit message, the protocol does not need to log acknowledgments.) Message *m3* is an in-transit message, and it is included in the checkpoint of process *P2*. Since process *P1* is storing its state when *m3* arrives, the acknowledgment is sent after the checkpoint is completed.

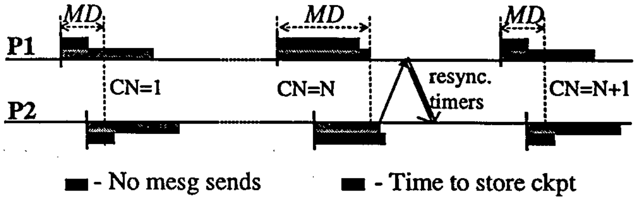

Figure 6. Several checkpoints.

The example from Figure 6 illustrates the behavior of the protocols throughout a long period of time. When the application starts, timers are well synchronized, and *ED* is much smaller than the time to save a checkpoint. As the application continues its execution, the value of *ED* increases because of the clock drifts. On the *N*'s checkpoint, the value of

Table 1. Applications used in the experiments.

Application	Problem Description
ga	1600 individuals, 10 million function evaluations
ising	1500 iterations on a 2500x2500 particle surface
povray	400x400 pixels, music.pov image

ED becomes larger than the time to store the checkpoint of process *P2*. After saving its state, process *P2* sends to process *P1* a request for timers' re-synchronization (in the example, *P1* is the coordinator). When processes create their next checkpoints, the value *ED* is again small.

6. Experimental Results

6.1. Application and Implementation

Three compute-intensive parallel applications were used in the experiments. These applications are complete programs, each with different characteristics in terms of frequency of communication and amount of information exchanged. Table 1 presents the inputs that were used for each application.

- ga: is a parallel implementation of the genetic algorithm system GENESIS 5.0 [15]. In the experiments, ga was used to solve a non-linear optimization problem. A node executes the genetic algorithm on its individuals and, after a few generations, sends a few individuals to one node and receives a number of individuals from another node.

- ising: is a parallel simulation model of physical systems such as alloys and polymers [17]. ising simulates, in two dimensions, the spin changes of Spinglass particles at different temperatures. In each step, a node computes the spin values of a subregion of the total particle surface and then exchanges the boundary particles with two other nodes.

- povray: is a parallel implementation of the raytracer POVRAY 2.2 [16]. This application uses a master-slave programming model. The main responsibility of the master is to distribute pixels of the image to the slaves. The slaves repeat the following steps: receive a number of pixels, calculate the color of the pixels, and return the results to the master.

The checkpoint protocol was implemented on the RENEW run-time system [14]. This system was designed for parallel programming in clusters of workstations, and it offers the standard interface *MPI - Message Passing Interface* [6]. Processes in RENEW communicate by exchanging datagrams over the UDP transport protocol. To ensure ordered and reliable message deliveries, RENEW utilizes acknowledgments and sequence numbers (as explained in Section 3). The implementation of the protocol was based in the procedures presented in Figures 2 and 4. The values of the maximum clock drift rate and minimum message delivery time were set to 10^{-5} and 0, respectively. The reader should notice that setting td_{min} equal to zero is a conservative assumption since it increases the size of critical interval. The value of D was set to 10 ms and the checkpoint period to 5 minutes.

The experiments were performed on a cluster of four Sun UltraSparc workstations running the Solaris 2.5 operating system. Each machine had 512 MBytes of main memory and 4 GBytes of local disk. The interconnection network was an 155 Mbits/s ATM. The processes' checkpoints were either saved in the local disk or in a remote HP workstation (also connected to the ATM network). All experiments were done during the night when the load in the network and machines was light.

6.2. Performance Results

Table 2 displays the experimental results obtained during the execution of the applications. The execution times without the checkpoint protocol are presented in the second column. In the column checkpoint is shown the number of checkpoints that were created, the size, and the average elapsed time to store a checkpoint in the local and remote disks. There are two checkpoint sizes for the ga and povray applications. The process that starts the ga application has a larger checkpoint size than the other processes because it allocates some extra data structures during initialization. The master of the povray application does not have to parse the image description file resulting in a smaller checkpoint size. The checkpoint storage time values correspond to execution of the instruction saveProcessState. As expected, it takes more time to write the checkpoints in a remote disk than in a local disk. However, since the network is fast and only a small number of processes execute concurrently, the difference between the two write times is not too large (it

Table 2. Performance Results on a Cluster of Workstations.

	No Ckp	Checkpoint				Time-Based			
				Local	Remote	Local		Remote	
	sec	#	KBytes	sec	sec	sec	%	sec	%
ga	2872	9	858/681	0.2/0.3	0.7/0.8	2972	3.5	2904	1.1
ising	3198	10	6828	1.5	3.3	3224	0.8	3232	1.1
povray	3091	10	683/22715	0.2/4.5	0.8/11.1	3120	0.9	3156	2.1

doubles). It is possible to observe in all cases that the overhead introduced by the checkpoint protocol is small, less than 3.5%. During the failure-free operation, the protocol only needs to create process checkpoints periodically, and to adjust the timers. All the other overheads were removed from the protocol.

The time-based protocol completes the creation of a new application checkpoint as soon as all processes save their states. The checkpoint latency is upper bounded by $D + 2\rho nT + \delta + tc_{max}$, where tc_{max} is the maximum time to store a process state. Since the protocol re-synchronizes the timers whenever $D+2\rho nT-td_{min}$ becomes larger than the minimum time necessary to create a checkpoint, tc_{min}, the checkpoint latency is in the order of $tc_{min}+tc_{max}+\delta$. For example, for `ising` with checkpoints created in the local disk and δ equal to one second, the latency is around four seconds.

Typical protocols that use message coordination need at least to exchange one round of messages to initiate the checkpoints (usually there are more rounds). The checkpoint latency for these protocols is in the order of $td_{max} + tc_{max}$, where td_{max} is the maximum time to deliver a message. The value of td_{max} varies depending on the system where the application is being run, but it can be seen as the maximum time the communication protocol is willing to wait to deliver a message, before it considers that there was a failure in the network or remote host. For example, in the implementation of TCP of Solaris 2.5 this value is more than 8 minutes. Therefore, in this environment, the checkpoint latency of this protocol is much larger than the time-based protocol.

In another paper, we have studied the performance of a coordinated protocol under failure conditions [14]. It was observed that processes could be restarted in a few seconds, and then they would continue their execution. Since during recovery the protocol lets the application run without interference, the time to restore the application to the prefailure state was approximately equal to the failure-free execution time. We expect to observe similar behavior for the protocol described in this paper.

7. Conclusions

In this paper we have described a new time-based checkpoint protocol that is able to save consistent and recoverable global states with small overheads. The protocol uses time to indirectly coordinate the checkpoint creation, avoiding extra message exchanges and message tagging. To guarantee that rapid recoveries can be accomplished, the protocol has checkpoint latencies independent of the message delivery times. By removing the assumption of small bounded delivery times, the protocol can be used in most computer systems, including systems connected by ethernet or ATM. The checkpoint protocol was implemented and evaluated using three parallel applications. Our results show that the protocol introduces performance costs smaller than 3.5%, both when checkpoints are saved in the local or remote disks.

References

[1] L. Alvisi, B. Hoppe, and K. Marzullo. Nonblocking and orphan-free message logging protocols. In *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing*, pages 145–154, June 1993.

[2] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[3] F. Cristian and C. Fetzer. Probabilistic internal clock synchronization. In *Proceedings of the 13th Symposium on Reliable Distributed Systems*, pages 22–31, October 1994.

[4] F. Cristian and F. Jahanian. A timestamp-based checkpointing protocol for long-lived distributed computations. In *Proceedings of the 10th Symposium on Reliable Distributed Systems*, pages 12–20, September 1991.

[5] L. S. DeBrunner and P. Ghali. An integrated error-recovery scheme for multicomputers. In *Proceedings of the Twenty-Seventh Asilomar Conference on Signals, Systems and Computers*, pages 817–821, November 1993.

[6] M. P. I. Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications*, 8(3/4), 1994.

[7] Y. Huang and Y.-M. Wang. Why optimistic message logging has not been used in telecommunications systems. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing*, pages 459–463, June 1995.

[8] D. B. Johnson and W. Zwaenepoel. Sender-based message logging. In *Proceedings of the 17th International Symposium on Fault-Tolerant Computing*, pages 14–19, July 1987.

[9] J. L. Kim and T. Park. An efficient protocol for checkpointing recovery in distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 4(8):231–240, August 1993.

[10] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, January 1987.

[11] K. Li, J. F. Naughton, and J. S. Plank. An efficient checkpointing method for multicomputers with wormhole routing. *International Journal of Parallel Programming*, 20(3):23–31, 1991.

[12] N. Neves and W. K. Fuchs. Using time to improve the performance of coordinated checkpointing. In *Proceedings of the International Computer Performance & Dependability Symposium*, pages 282–291, September 1996.

[13] N. Neves and W. K. Fuchs. Fault detection using hints from the socket layer. In *Proceedings of the 16th Symposium on Reliable Distributed Systems*, pages 64–71, October 1997.

[14] N. Neves and W. K. Fuchs. RENEW: A tool for fast and efficient implementation of checkpoint protocols. In *Proceedings of the 28th International Symposium on Fault-Tolerant Computing*, June 1998.

[15] N. Neves, A.-T. Nguyen, and E. L. Torres. A study of a non-linear optimization problem using a distributed genetic algorithm. In *Proceedings of the International Conference on Parallel Processing*, volume II, pages 29–36, August 1996.

[16] POV-Ray Team. *Persistency of vision ray tracer (POV-Ray): User's Documentation*, 1993. http://povray.org.

[17] J. G. Silva, J. Carreira, H. Madeira, D. Costa, and F. Moreira. Experimental assessment of parallel systems. In *Proceedings of the 26th International Symposium on Fault-Tolerant Computing*, pages 415–424, June 1996.

[18] L. M. Silva and J. G. Silva. Global checkpointing for distributed programs. In *Proceedings of the 11th Symposium on Reliable Distributed Systems*, pages 155–162, October 1992.

[19] L. M. Silva and J. G. Silva. On the optimum recovery of distributed systems. In *Proceedings of the EUROMICRO Conference*, pages 704–711, September 1994.

[20] A. P. Sistla and J. L. Welch. Efficient distributed recovery using message logging. In *Proceedings of the Eighth Annual Symposium on Principles of Distributed Computing*, pages 223–238, August 1989.

[21] S. W. Smith, D. B. Johnson, and J. D. Tygar. Completely asynchronous optimistic recovery with minimal rollbacks. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing*, pages 361–370, June 1995.

[22] R. E. Strom, D. F. Bacon, and S. A. Yemini. Volatile logging in n-fault-tolerant distributed systems. *Proceedings of the 18th International Symposium on Fault-Tolerant Computing*, pages 44–49, June 1988.

[23] R. E. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, August 1985.

[24] Y. Tamir and T. M. Frazier. Application-transparent process-level error recovery for multicomputers. In *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Services*, pages 296–305, January 1989.

[25] Y. Tamir and C. H. Sèquin. Error recovery in multicomputers using global checkpoints. In *Proceedings of the International Conference on Parallel Processing*, pages 32–41, August 1984.

[26] Z. Tong, R. Y. Kain, and W. T. Tsai. A low overhead checkpointing and rollback recovery scheme for distributed systems. In *Proceedings of the 8th Symposium on Reliable Distributed Systems*, pages 12–20, October 1989.

[27] N. H. Vaidya. On checkpoint latency. In *Proceedings of the Pacific Rim International Symposium on Fault-Tolerant Systems*, pages 60–65, December 1995.

[28] Y.-M. Wang and W. K. Fuchs. Lazy checkpoint coordination for bounding rollback propagation. In *Proceedings of the 12th Symposium on Reliable Distributed Systems*, pages 86–95, October 1993.